Semantic Web Research Trends and Directions

Jennifer Golbeck[1], Bernardo Cuenca Grau, Christian Halaschek-Wiener, Aditya Kalyanpur, Yarden Katz, Bijan Parsia, Andrew Schain, Evren Sirin, and James Hendler

MINDSWAP, University of Maryland, College Park, MD 20742, USA,
golbeck@cs.umd.edu, bernardo.cuenca@uv.es, {halasche,aditya}@cs.umd.edu,
yarden@umd.edu, bparsia@isr.umd.edu,
andrew@schain.org,{evren,hendler}@cs.umd.edu
WWW home page: http://www.mindswap.org

Abstract. The Semantic Web is not a single technology, but rather a collection of technologies designed to work together. As a result, research on the Semantic Web intends both to advance individual technologies as well as to integrate them and take advantage of the result. In this paper we present new work on many layers of the Semantic Web, including content generation, web services, e-connections, and trust.

1 Introduction

Defining the Semantic Web is a difficult task. It is the next generation of the web. It is a set of languages and standards. It has a strong logic and reasoning component. Web services, web portals, markup tools, and applications are all components. As a result, there are many interesting, intertwined research areas.

In this paper, we address several emerging trends of research in the Semantic Web space. We begin by presenting two tools for creating content on the Semantic Web: SWOOP, an ontology browser and editor, and PhotoStuff, an image annotation tool that integrates with a web portal. We follow with descriptions of E-Connections, a logical formalism for combining ontologies, and of work in web services. Finally, we describe a project using social trust on the semantic web that builds upon the previous work to create end user applications that benefit from the semantic foundation.

2 Swoop - Web Ontology Editor

Swoop is a hypermedia-inspired Ontology Browser and Editor based on OWL, the first standardized Web-oriented ontology language. Swoop takes the standard Web browser as the UI paradigm, believing that URIs are central to the understanding and construction of OWL Ontologies. The familiar look and feel of a browser emphasized by the address bar and history buttons, navigation side bar, bookmarks, hypertextual navigation etc are all supported for web ontologies, corresponding with the mental model people have of URI-based web tools based on their current Web browsers.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Semantic Web Research Trends and Directions	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Maryland,MINDSWAP,College Park,MD,20742**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**24**	

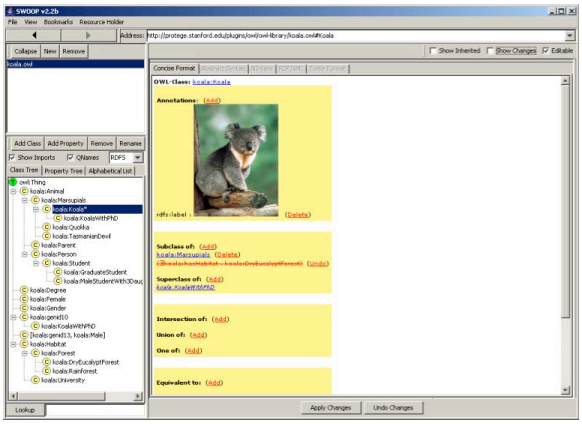

Fig. 1. The Swoop Interface

All design decisions are in keeping with the OWL nature and specifications. Thus, multiple ontologies are supported easily, various OWL presentation syntax are used to render ontologies, open-world semantics are assumed while editing and OWL reasoners can be integrated for consistency checking. A key point in our work is that the hypermedia basis of the UI is exposed in virtually every aspect of ontology engineering — easy navigation of OWL entities, comparing and editing related entities, search and cross referencing, multimedia support for annotation, etc. — thus allowing ontology developers to think of OWL as just another Web format, and thereby take advantage of its Web-based features.

2.1 Summary of Features

Swoop functionality is characterized by the following basic features (for an elaborate discussion of the features see [**?**]).

- Multiple Ontology Browsing and Editing Swoop has a variety of mechanisms for pulling in different Web ontologies into its model using bookmarks; loading via the address bar; during navigation across ontologies etc. Additionally, ontology browsing and editing are done in a single pane, which helps to maintain context.
- Renderer Plugins for OWL Presentation Syntaxes Swoop bundles in six renderers; two Ontology Renderers-Information and Species Validation; and four Entity Renderers Concise Format, OWL Abstract Syntax, Turtle and RDF/XML. By supporting different presentation syntaxes, accessibility is enhanced tremendously.
- Reasoner plugins in Swoop Swoop contains two additional reasoners (besides the basic Reasoner that simply uses the asserted structure of the ontology):

RDFS-like and Pellet [2]. While the former is a lightweight reasoner based on RDFS semantics, the latter, Pellet, is a powerful description logic tableaux reasoner. The reasoners provide a tradeoff between speed and quality of inference results, e.g., the RDFS-like reasoner, while much faster than Pellet in execution, is unsound (results maybe inaccurate if the ontology is inconsistent) and incomplete (does not list all possible inferences). Yet, in most cases, it provides interesting and useful results for ontology authors, and moreover, the reasoners can be used in conjunction to analyze the ontology quickly while editing it.

- Semantic Search Swoop takes inspiration from the hyperlink based search and cross-referencing utility present in a programming IDE such as Eclipse (http://www.eclipse.org/). For example, a non-standard search feature in Swoop is Show References, which lists all references of a single OWL entity (concept/property/individual) in local or external ontological definitions. This feature can help understand usage of an entity in a specific context and is especially useful in ontology debugging where non-local interactions can lead to errors of a single concept.
- Ontology Versioning Swoop supports ad hoc undo/redo of changes (with logging) coupled with the ability to checkpoint and archive different ontology versions. Each change set or checkpoint can be saved at three different granularity levels - entity, ontology, workspace - which specify its scope. While the change logs can be used to explicitly track the evolution of an ontology, checkpoints allows the user to switch between versions directly exploring different modeling alternatives.Swoop also has the option to save checkpoints automatically, i.e., during specific tasks such as loading a new ontology, applying changes, removing or renaming an entity etc. Note that each time a checkpoint is saved, a snapshot of the entity definition is cached as well, and can be used to preview a checkpoint before reverting back to it.

2.2 Advanced Features

Additionally, Swoop has the following advanced features, which represent work in progress:

- Resource Holder (Comparator/Mapping utility) In Swoop, there is a provision to store and compare OWL entities during an extended search and browsing process via the Resource Holder panel. This common placeholder acts as an excellent platform for performing interesting engineering tasks such as comparing differences in definitions of a set of entities; determining semantic mappings between a specific pair of entities or simply storing entities for reusing in another ontology.
- Automatic Partitioning of OWL Ontologies (using E-Connections): Swoop has a provision for automatically partitioning OWL Ontologies by transforming them into an E-connection. For more details on the theory and significance of E-connections, their use in the context of OWL Ontologies see section 4.

– Ontology Debugging and Repair Swoop uses two techniques for ontology debugging and repair: glass box and black box. In glass box techniques, information from the internals of the reasoner is extracted and presented to the user (typically used to pinpoint the type of clash/contradiction and axioms leading to the clash [4]). In black box techniques, the reasoner is used as an oracle for a certain set of questions e.g., the standard description logic inferences (subsumption, satisfiability, etc.) and the asserted structure of the ontology is used to help isolate the source of the problems (can be used to find dependencies between unsatisfiable classes).
– Annotea Client in Swoop: Collaborative Annotations and Change Sets

The Annotea protocol [5] allows for publishing and finding out-of-band annotations on arbitrary web resources. Annotea support in Swoop is provided via a simple plug in that uses the default Annotea RDF schema to specify annotations. Any public Annotea Server can then be used to publish and distribute the annotations created in Swoop. In addition to annotations, users can attach and share Ontology Change sets that are created using Swoop.

3 PhotoStuff Semantic Image Annotation Tool

PhotoStuff is a platform-independent image annotation tool that allows users to annotate regions of an image with respect to concepts in any ontology specified in RDFS or OWL. It provides the functionality to import images (and their embedded metadata), ontologies, instance-bases, perform markup, and export the resulting annotations to disk or a Semantic Web portal.

3.1 Overview

PhotoStuff is designed to load multiple ontologies at once, enabling a user to markup images with concepts distributed across any of the loaded ontologies.

this was all cut and used to create the paragraph that follows the commented out text

Each ontology is stored in a local knowledge base. The ontologies are visualized in both a class tree and list (depicted below in Figure 2 in the far left pane of the tool). User can load images (from the Web and/or local disk) in PhotoStuff. The tool currently takes advantage of existing embedded image metadata by extracting and encoding this information into RDF/XML, thus allowing embedded metadata to be directly incorporated into the tool and the Semantic Web in general. Using a variety of region drawing tools, users are able to highlight regions around portions of loaded in PhotoStuff. The terms listed in both the tree and list can be dragged into any region, or into the image itself, creating a new instance of the selected class. An instance creation form is dynamically generated from the properties of the selected class (range restrictions are imposed). Especially valuable, existing instances can be loaded from any URI on the Web. Using these preloaded instances, depictions can reference existing instances.

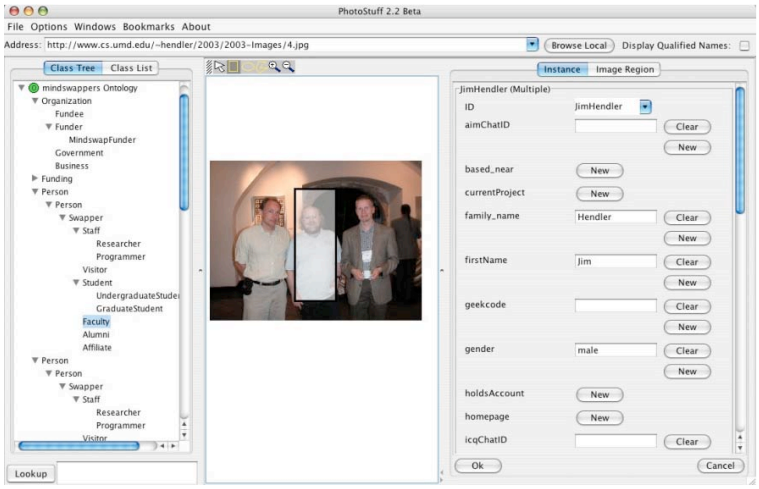

Fig. 2. PhotoStuff Screenshot

PhotoStuffs underlying functionality of making assertions regarding the high level content of digital images is driven by an ontology-based approach. More specifically, an image-region ontology is used to provide the expressiveness required to assert what is actually depicted within an image, as well information about the image (date created, etc.). In this work, such an ontology has been specified, using OWL, which defines a small set of concepts for images, videos, regions, depictions, etc. Using these concepts and their associated properties, it is possible to assert that an image/imageRegion depicts some instance, etc. The complete RDF/XML representation of the ontology described here is available on the MINDSWAP website .

The ontologies are visualized in both a class tree and list (depicted below in Figure 2 in the far left pane of the tool). User can load images (from the Web and/or local disk) in PhotoStuff. The terms listed in both the tree and list can be dragged into any region, or into the image itself, creating a new instance of the selected class. An instance creation form is dynamically generated from the properties of the selected class (range restrictions are imposed). Especially valuable, existing instances can be loaded from any URI on the Web. Using these preloaded instances, depictions can reference existing instances.

Communication between the two is done via HTTP/S using simple Web server side Python scripts and other available technology, including WebDAV.

PhotoStuff maintains a loose coupling with a Semantic Web portal There are three ways in which PhotoStuff interacts with the portal, namely retrieving all instances that have been submitted to the portal, submitting generated RDF/XML, and uploading local images so they can be referenced by a URI (thus allowing them to be referenced using RDF/XML).

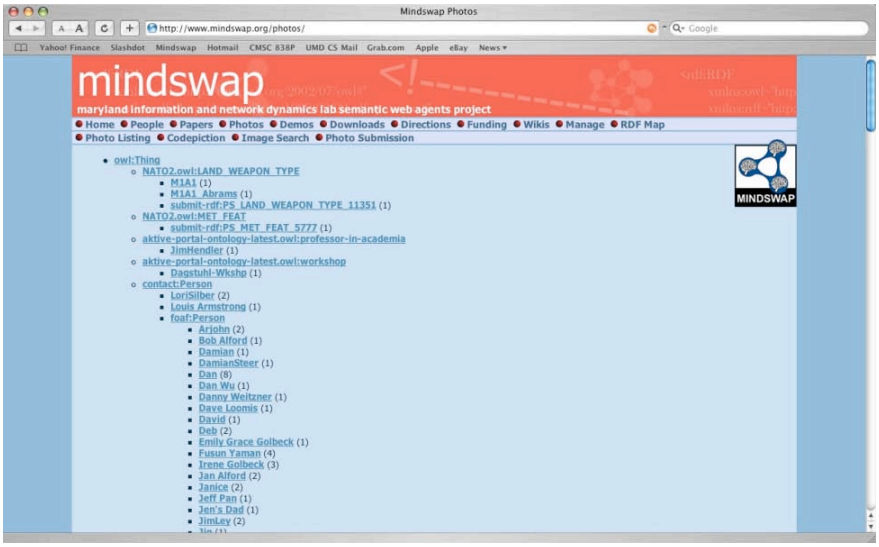

Fig. 3. Image Browsing By Class

The ability to retrieve markup from the portal allows users to annotate photos against existing instances they, or others, have already created. Essentially in PhotoStuff, URIs (to RDF/XML documents containing instances) can be pre-specified from which the tool can read in. These instances are read into the tools local knowledge base and subsequently available for annotation purposes.

When submitting newly created image annotations to the portal, PhotoStuff can add to existing instance bases already maintained on the portal and make it available on the Semantic Web. Lastly, PhotoStuff provides the functionality to upload any local image that the user has annotated. When a user imports a local image, a copy of the image is uploaded to a WebDAV repository maintained from with the portal. This is required so that the annotations can reference the image on the Web.

3.2 Image Metadata Browsing and Searching

After metadata of annotated images is submitted to the Semantic Web portal, semantics based image browsing and searching is provided. The portal uses information from the various ontologies image content is annotated against to guide the display of and interaction with metadata (e.g., class hierarchy drives the browsing layout, as seen in Figure 3). The main interface for browsing images is displayed below in Figure 3. All instances that are depicted within an image are presented. As noted earlier, the underlying class of each instance drives the actual order of the depictions, thus providing a high level view of all the metadata of images that have been annotated using PhotoStuff.

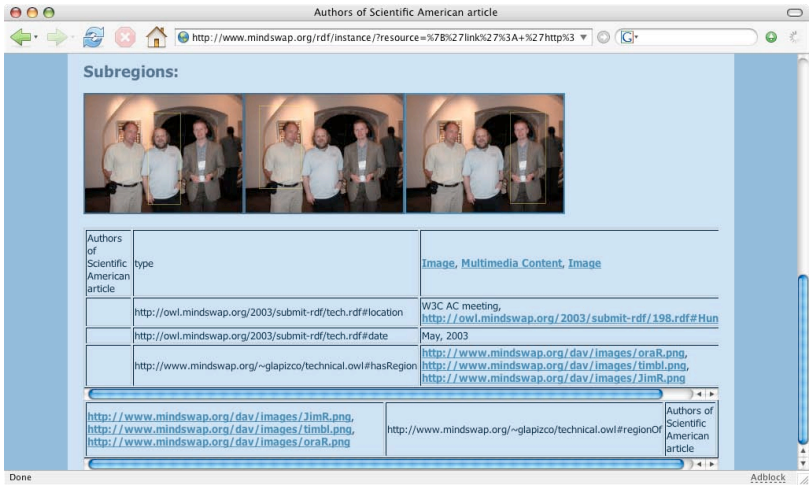

Fig. 4. Instance Descriptions and Image Co-Region Browsing

When an instance is selected, the user is presented with all images in which the instance is depicted (illustrated in Figure 4). All of the metadata regarding that instance is presented to the user as well (currently in tabular form). We note here that this links the images with pre-existing metadata maintained in the portal.

In Figure 4, it can be seen that specific regions are highlighted. This is accomplished using an SVG outline of the region drawn on the various images (this data is embedded in RDF/XML as well). By selecting an image region, the various co-regions of the selected image region are displayed (shown in Figure 4). This allows browsing of the metadata associated with the various regions depicted in the image. Lastly, the portal provides support for searching image metadata. Images are searchable at the instance and class level.

4 E-Connections of Web Ontologies

An E-Connection is a knowledge representation (KR) formalism defined as a combination of other logical formalisms. The component logics that can be used in an E-Connection include Description Logics (and hence OWL-DL), some temporal and spatial logics, Modal and Epistemic logics. E-Connections were originally introduced as a way to go beyond the expressivity of each of the component logics, while preserving the decidability of the reasoning services. Obviously, different component logics will give rise to different combined languages, with different expressivity and computational properties.

In a Semantic Web context, E-Connections allow the user to combine OWL-DL ontologies. A combination of OWL-DL ontologies is called a combined knowledge base. A combined knowledge base is a set of "connected ontologies. These

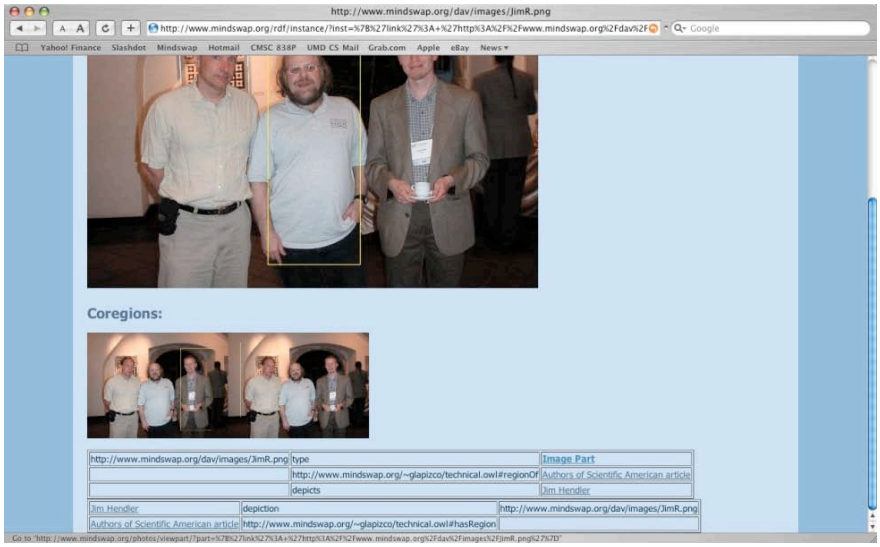

Fig. 5. Image Co-Region Browsing

connected ontologies are basically OWL-DL ontologies extended with the ability to define and use link properties.

In OWL-DL, object properties are used to relate individuals within a given ontology, while datatype properties relate individuals to data values. In a combined knowledge base, link properties are used to relate individuals belonging to different ontologies in the combination.

A link property can be used to define classes in a certain ontology in terms of other classes corresponding to different ontologies in the combination. For example, a "Graduate Student" in an ontology about "people" could be defined as a student who is enrolled in at least one graduate course, by using the class "Student" in the people ontology and a someValuesFrom restriction on the link property "enrolledIn" with value "GraduateCourse": a class in a different ontology dealing with the domain of "academic courses".

Link properties are logically interpreted as binary relations, where the first element belongs to the "source" ontology and the second to the "target ontology" of the link property. Conceptually, a link property will be defined and used in its "source" ontology. For example, the link property "enrolledIn" would be defined as a link property in the "people" ontology with target ontology "academic courses".

From the modeling perspective, each of the component ontologies in an E-Connection is modeling a different application domain, while the E-Connection itself is modelling the union of all these domains. For example, an E-Connection could be used to model all the relevant information referred to a certain university, and each of its component ontologies could model, respectively, the domain

of people involved in the university, the domain of schools and departments, the domain of courses, etc.

As an example, suppose that we want to model the domain of "tourism" as a combination of the following sub-domains: "travel accommodations", "leisure activities", "travel destinations", and "people". We want to model each application sub-domain in a different ontology and then use link properties to represent their relationships.

We would like to define classes like "BudgetDestination (a travel destination which provides a choice of budget accommodations) in the "destinations" ontology, or "CaribbeanHotel (a hotel accommodation offered at a Caribbean destination) in the "accommodations" ontology.

For such a purpose, we define the link properties "providesAccomodation and "offersActivity, which relate the domain of "destinations" to the domain of "accommodations" and "activities" respectively, and the link property "hasHobby, that relates the domain of "people" and the domain of "activities".

Restrictions on link properties can be used to generate new classes. For example, we can define a "budget destination" in the destinations ontology as a travel destination that offers at least one kind of budget accommodation.

4.1 Applications

Integration of OWL Ontologies E-Connections can be used as a suitable representation for combining and integrating OWL-DL ontologies on the Semantic Web. For example, suppose a set of ontologies that have been independently developed and are now required to interoperate within an application. In order to provide support for integrating Web ontologies, OWL defines the owl:imports construct, which allows to include by reference in a knowledge base the axioms contained in another ontology, published somewhere on the Web and identified by a global name (a URI). However, the only way that the owl:imports construct provides for using concepts from a different ontology is to bring into the original ontology all the axioms of the imported one. This keeps the ontologies in separate files, providing some syntactic modularity, but not a logical modularity, as in the case of E-Connections.

Analysis and Decomposition of OWL Ontologies E-Connections have also proved useful for analyzing the structure of knowledge bases and, in particular, to discover relevant sub-parts of ontologies, commonly called modules. For example, suppose one wants to annotate a word in a document using a class from an ontology, published elsewhere on the Web. It would be natural for a Semantic Web application to retrieve only the minimal set of axioms that "capture" the meaning of that specific class in the remote ontology.

In general, the ability to identify relevant sub-parts of ontologies is important for virtually every Semantic Web application. For example, in ontology engineering, achieving some sort of "modularity" is a key requirement to facilitate collaborative, large scale, long term development. Modularity worthy of

the name should provide benefits in the following aspects: processability of the ontology by applications, evolvability and maintenance, knowledge reuse and understandability for humans.

E-Connections provide a sensible way for identifying modules within an ontology. The main idea of the method is to transform the original ontology into an E-Connection with the largest possible number of connected knowledge bases such that it preserves the semantics of the original ontology in a very specific way. The obtained connected ontologies are used to generate, for each entity, a subset of the original ontology that encapsulates the entity, i.e., that preserves a certain set of crucial entailments related to the entity. The key advantage of relying on E-Connections for such a task is that modules are obtained in a completely automatic way, without relying on any heuristics or user intervention.

Integration of OWL Ontologies with Spatial and Temporal Knowledge Bases Spatial and temporal information is crucial for many Semantic Web applications, such as Geographic Information Systems and Web Services. However, OWL has not been conceived for such a purpose and hence representing spatial and temporal knowledge in OWL is always hard and sometimes not even possible.

E-Connections can be used to connect Description Logics knowledge bases (and hence OWL-DL ontologies) with many spatial and temporal modal logics. Hence, E-Connections provide an infrastructure for providing interoperability between OWL ontologies and spatial and temporal knowledge bases.

Tool Support We have provided support for E-Connection in both an OWL ontology editor, SWOOP, and an OWL reasoner, Pellet. Our aim has been to build an E-Connection aware infrastructure that people with a large commitment to OWL will find understandable and useful. We have been mostly concerned with reusing as much of our current tool support for OWL as possible. Our experience in implementing OWL tools has taught us that implementing a SemanticWeb Knowledge Representation formalism is different from implementing the very same formalism outside the Semantic Web framework. In particular, any new formalism for the Semantic Web needs to be compatible with the existing standards, such as RDF and OWL. For such a purpose, we have extended the syntax and semantics of the OWL-DL recommendation. Thus, we have explored both the issues related to the implementation of a new KR formalism, E-Connections, and those concerning its integration in a Semantic Web context.

5 Web Services

In this section, we describe various different projects and applications we developed for Web Services. Our work on Web Services concentrates on three core tasks: matchmaking, composition, and execution; always focusing on the end-to-end experience. To enable the automation of such tasks, we needed expressive

descriptions of Web Services. For this reason, we have been involved with the development of the OWL-S language. The OWL-S is a collection of ontologies written in OWL to describe Web Services. Currently it is the most mature and probably the most widely deployed comprehensive Semantic Web Service technology.

Initially, we have looked at the problem of semi-automated composition of services [**?**][**?**], wherein the composition is primarily driven by a person. Building complex workflows by hand is notoriously difficult. At every step of a composition, users face a plethora of choices. Our assisted composition approach uses the richness of Semantic Web Service descriptions and information from the compositional context to filter matching services and help select appropriate services.

Semi-automated composition is a goal-directed approach where the composition is gradually generated with a forward or backward chaining of services. At each step, a new service is added to the composition and further possibilities are filtered based on the current context and user decisions. The system primarily uses the input/output types of the services to find compatible services. Subsumption relation between OWL concepts is inspected to find flexible matches between services. The notion of ontology mapping is used to improve the interoperability between services described using ontologies that contains fairly similar but distinct concepts. OWL aims to increase the interoperability by allowing full and partial mappings between distributed ontologies using logical axioms. However, in some cases it is not possible to map two terms without ad hoc procedural computations. In our system, such procedural mappings are also described as Web Services (where translation is achieved using XSLT transformations) and these services are classified in a special service category. Then, system can automatically fuse these translation services to world affecting actions to produce more and better matches for a composition step.

Finding compatible services does not provide enough clues to a domain expert as to which service is the most suitable choice to accomplish the task at hand. There are generally a large number of type-compatible services with different characteristics. To filter these possibilities, we utilize the information about other service parameters that describe the non-functional attributes, i.e. properties other than the input/output functional signature. The non-functional attributes of a service include the information about the service provider, the QoS information about the service reliability and cost, the security and privacy policies the service adheres to and so on. Constraints on these properties may be used to filter the compatible services to find the most relevant service for the current task. Based on the ontologies used to describe such non-functional attributes, we automatically generate filtering forms where user can enter such constraints easily.

The work described above was done in collaboration with Fujitsu Labs of America, College Park and most of these features made it way into, or inspired aspect of, Fujitsu's Task Computing Environment [3]. In a joint work, we also produced a semi-automated ontology mapping tool called OntoLink [4] to ease the generation of translation services.

The OWL-S descriptions are invaluable for many web service tasks. However, the intended semantics of OWL-S service descriptions is not expressed (or expressable, often) in OWL. Furthermore, working with OWL-S descriptions at the RDF or even the OWL level is quite difficult and tedious as they tend to be at the wrong level of abstraction. For this reason, we have developed the Mindswap OWL-S API [**?**], a Java API developed for parsing, validating, manipulating, executing, matching, and in general reasoning over OWL-S descriptions. OWL-S API provides a programmatic interface that has been designed closely to match the definitions in the OWL-S ontology. This interface helps developers build applications using OWL-S without having to deal with all the details of the syntax. The API provides an execution engine to invoke atomic services as well as composite processes that are built using the OWL-S control constructs.

Our work on Semantic Web Services focuses on the automated composition of services using Hierarchical Task Network (HTN) planning formalism [**?**]. Instead of looking at each service in isolation, we focus on the descriptions of workflow templates. Workflow templates are used for various different tasks such as encoding business rules in a B2B application, specifying domain knowledge in a scientific Grid application, and defining preferences for users that interact with Web Services. A template describes the general outline of how to achieve a certain task and the planners job is to find which of these possible execution paths will be successful in the current state of the world with the current requirements and preferences.

We have developed a mapping [**?**] from OWL-S to the HTN formalism as implemented in the SHOP2 planner [**?**]. Using this translation, we implemented a system that plans over sets of OWL-S descriptions using SHOP2 and executes the resulting plans over the Web. The planning system is also capable of executing information-providing Web Services during the planning process to decide which world-altering services to use in the plan. It is possible to completely automate the information gathering process by inspecting the preconditions of a service and finding the relevant information-providing services during planning [**?**].

The problem of using a planner for composing OWL-S services is the limited reasoning capabilities provided by the planner. The typical logic for expressing preconditions and effects in a planning system has a radically different expressiveness then RDF and OWL do. In order to evaluate such formulas, the planners must understand the semantics of OWL. We have integrated the SHOP2 planner with our OWL-DL reasoner Pellet to overcome this problem [**?**]. In this integrated system, the precondition evaluation is done by the OWL reasoner against the local and remote OWL ontologies. Such integration poses some efficiency challenges because the planner constantly simulates the effects of services resulting in modifications to its internal OWL KB and continues to query the KB as it continues the planning process. We have developed several query optimization methods to increase the performance of the system.

Most recently, we are looking at how to handle more expressive workflow templates where HTN task selection is extended to incorporate OWL-S based matchmaking [**?**]. We aim to extend the OWL-S language to describe the ab-

stract processes using profile hierarchies and complex OWL concept descriptions. This extension will allow a clearer way to define soft and hard preferences in a template along with a prioritization of these preferences. Consequently, we can have more elaborate ranking mechanisms for choosing out of the possible options during planning. We are currently working on the implementation of this extended HTN formalism (HTN-DL) [**?**].

6 Combining OWL and Rule Languages

6.1 Introduction

The need for integrating Datalog languages within the Semantic Web DL ontology languages has been the focus of several research initiatives [**?**,**?**,**?**,**?**,**?**,**?**]. The combination seems to take advantage of the best characteristics of both formalisms : DLs are appropriate for structuring knowledge in terms of concepts and relationships, but the subset of Horn rules that can be expressed in a decidable DL are only those that satisfy the tree-like property. In Datalog we can express any *safe* Horn rule where the interaction between variables is unrestricted.
On the other hand, in DL, an axiom variable may be either existentially or universally quantified and we can express disjunction and disjointness of classes; therefore, complex class expressions and relationships required in many knowledge domains can be expressed. In plain Datalog, disjunction and disjointness cannot be expressed, and all variables are universally quantified.
With respect to query languages, OWL-DL in general provides ABox query languages which support instantiation (if an individual is an instance of a class), realisation (most specific classes that an individual belongs to) and retrieval (instances of a certain class). Techniques have been developed that are used to provide a more expressive query language that involves classes, roles and variables [**?**,**?**]. The basic idea for queries involving roles is to convert the role term in a query to a class term; this technique is called 'rolling up' a query. It is defined for tree-shaped query graphs. When combining DL and Rules languages, Datalog is used as a query language. subsectionApproaches There are three main approaches for integrating OWL and rule languages:

1. Intersection
   This is the approach taken by Description Logic Programs (DLP) [**?**], which finds the expressive intersection between DL and Logic Programming (LP) without function symbols. Some DL constructors cannot be mapped, and DL queries are reduced to LP queries. This is not a practical interoperability solution for rules and OWL for two reasons:
   - If the DLP subset is used, then most of the power and naturalness of expression of both languages is sacrificed.
   - If a DLP ontology is used as a common core to be extended, as needed, by OWL on the one hand and a rules language on the other, then one ends up with two, incompatible ontologies. The extended ontologies are completely insensitive to the other's semantics.

2. Hybrid systems
   These systems consider a KB with two subsystems: the DL and the Datalog subsystem. They permit the use of "DL atoms" in the heads and bodies of Datalog style rules [**?**,**?**,**?**,**?**]. All these combinations have the virtue of increasing the expressivity of the combined logics while remaining decidable. AL-Log, for example, involves the weakest combination. The interaction between the subsystems is done through the specification of constraints in the Datalog rules which "type" variables appearing elsewhere in the body These constraints are expressed in terms of $\mathcal{ALC}$ classes. The structural subsystem represents the knowledge in terms of classes, roles and individuals, while the relational subsystem allows to express constrained Horne clauses and the formulation of deductive queries in terms of these clauses.
   In CARIN [**?**], classes and roles may appear in the antecedent of Horn rules in the Datalog component. In the DL-SafeRules approach [**?**], classes and roles are allowed to occur in the antecedent and consequent of Horn rules in the Datalog component. Each variable in a rule is required to occur in a non-DL atom in the rule body. DL-safety ensures that each variable is bound to individuals that are explicit in the ABox.
3. Supersetting
   The final method of combining rules and OWL is to simply extend OWL axioms with fully generalized conditionals with arbitrary combinations of DL-like atoms on both sides of the rule. This is the approach taken by the Semantic Web Rules Language (SWRL) [**?**] when it includes Datalog safe rules without negation and with built-ins . SWRL has the disadvantage of being undecidable. However, it has been incorporated in other submissions to the W3C, in particular, OWL-S. SWRL is a proper superset of all the hybrid systems discussed above.

subsectionImplementation We attempt to evolve OWL DL toward SWRL, but cautiously and carefully. We strive to retain practical decidability. We seek to meet identifiable needs as simply as possible. We also attempt to make use of existing work on integrating rules systems, especially Datalog, with expressive description logics. We want to build a rules system that people with a large commitment to OWL will find understandable and useful. With such an aim, we explore different ways to give the users what they want, while staying "close" to OWL, and "cautiously far" from the full expressivity of SWRL. We have implemented a hybrid system for combining OWL and rules by extending our Web ontology browsing and editing tool, SWOOP (http://www.mindswap.org/2004/SWOOP). The hybrid system, as a decidable fragment of SWRL, provides validation of SWRL's usefulness and implementability. The features of our system are the following:

1. Rules Expressivity
   Following the SWRL proposal, the user can edit and browse rules. Different (increasing) "levels" of rules expressivity are considered: syntactic sugar for OWL, AL-log, CARIN, DL Safe and SWRL. For each set of rules, the number of rules in each level and the expressivity of the set is displayed.

2. Syntactic sugar for OWL
We define a method of representing a subset of SWRL directly in OWL, while preserving its semantics. We do this by utilizing techniques used in the processing of conjunctive ABox queries to transform rules into class axioms in DL. This approach allows for some rules to be treated as syntactic sugar for complex DL class expressions and axioms, but it imposes significant restrictions on the structure of these rules. The rolling-up technique used in answering conjunctive queries can also be applied to a subset of SWRL, to gain some of the syntactic expressivity of rules without extending the semantic expressivity of OWL-DL. In this approach, the antecedent and consequent of a rule are each treated as a conjunctive query, and transformed into DL class expressions using the rolling-up technique. The addition of the assertion that the antecedent class expression is a subclass of the consequent class expression ensures the intended rule semantics. We must define the set of rules (subset of SWRL) that can be translated directly into DL class axioms using this technique. Since the rolling-up technique is to be applied to both the antecedent and consequent, it is clear that each must satisfy the requirements of a conjunctive query, described above. However, further conditions must be imposed to ensure that the subclass relation between the two resulting classes will have its intended semantics. These conditions are the following, depending on the number of variables shared between the consequent and antecedent:
   - If 0 variables are shared, then the rule can be represented in OWL if at least one individual is shared. All variables in each query are undistinguished.
   - If 1 variable is shared, then the rule can be represented in OWL. Only the shared variable is distinguished, and so is used as the target of rolling-up.
   - If 2 or more variables are shared, then the rule can not be represented directly in OWL.

Consider the following rule, which describes a set of conditions that imply a given computer is a fast computer.
$FastComputer(?c) : -$
$Computer(?c), hasCPU(?c, ?cpu), hasSpeed(?cpu, ?sp), HighSpeed(?sp).$
The initial step, the transformation of the rule into two conjunctive queries, is straightforward. For the consequent and antecedent, each predicate in the rule maps directly to a conjunctive query term by the mapping below:
$C(?x) \rightarrow ?x : C$
$R(?x, ?y) \rightarrow \langle ?x, ?y \rangle : R$
Applying this mapping to the consequent of the example rule above simply yields $?c{:}FastComputer$. However, for the more complex antecedent, we obtain the following query:
$?c : Computer \wedge \langle ?c, ?cpu \rangle : hasCPU \wedge \langle ?cpu, ?sp \rangle : hasSpeed \wedge ?sp : HighSpeed$
Applying the rolling-up technique to each of the queries will yield the class expressions needed to represent the rule in DL. Both queries contain the variable $?c$, so this will be the only distinguished variable in each of the queries. Rolling-up the consequent produces the simple class expression $FastComputer$.

Rolling-up the antecedent query to this variable generates the class expression $Computer \sqcap \exists hasCPU.\exists hasSpeed.HighSpeed$. To complete the representation of the original rule, we simply need to add the following subclass axiom to the KB:
$Computer \sqcap \exists hasCPU.\exists hasSpeed.HighSpeed \sqsubseteq FastComputer$

3. Query answering for AL-log
We have implemented an $\mathcal{AL}$-Log reasoner. It computes answers to queries based on the specification of both components and is based on the notion of *constrained SLD-derivation* and *constrained SLD-refutation*, as presented in [**?**]. The system has been implemented in Prolog, coupled to our OWL reasoner Pellet (http://www.mindswap.org/2003/pellet). The key idea of this implementation is to pre-process all of the DL atoms that appear in the Datalog rules, and include them as facts in the relational subsystem. In order to cover all the possible models, if two or more rules have the same (obviously non-DL) atom in the head, and they also share a DL-atom in the body, whose (single) variable appears in the same argument of the head in the rules, then the disjunction of all those (unary) DL-atoms must also be computed and realized by the DL reasoner. Once the pre-processing is done, any query can be answered by the relational component using any of the known techniques for Datalog query evaluation.

7 Trust: Computations and Applications

One of the ultimate goals of the Semantic Web is to produce a so-called "Web of Trust". Much work in this space has been made in the spheres of security, authentication, and privacy. However, the social component of trust is one that is both important and ideally suited for the Semantic Web. When the Semantic Web-based social networks are augmented with trust information, it is possible to make computations over the values, and integrate the results into applications.

7.1 Adding Trust to FOAF

In the FOAF vocabulary, the only mechanism for describing social connections between people is the foaf:knows property which indicates one person knows another. The FOAF Trust Module extends the FOAF vocabulary by providing a mechanism for describing the trust relationships. It allows people to rate the trustworthiness of another person on a scale from 1 to 10 where 1 is low trust and 10 is a high trust. This scale is intuitive for human users, but our algorithms can be applied to any scale of values.

The Trust Project[1] regularly spiders the Semantic Web for trust files, and maintains a network with over 2,000 people. The ontology is also used by the FilmTrust social network [**?**] with over 400 members. These networks can be seen in Figure 6, and are used as testbeds for our algorithms for computing trust.

[1] see http://trust.mindswap.org

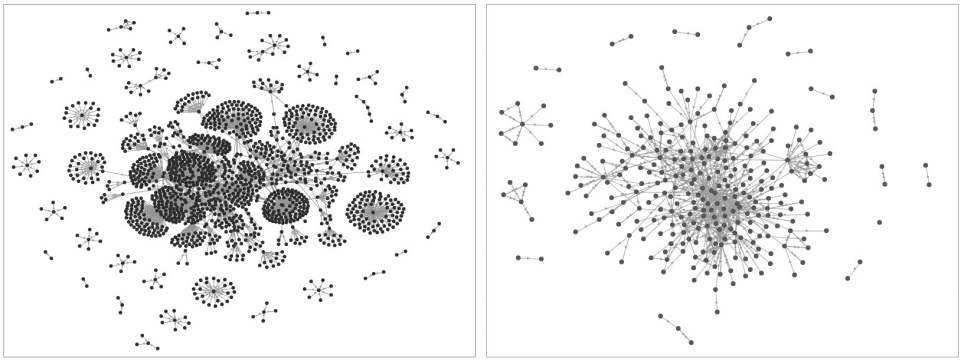

Fig. 6. Visualizations of the Trust Project's network (left) and the FilmTrust network (right).

7.2 Computing with Trust

If one person, the *source*, wants to know how much to trust another person, the *sink*, how can that information be obtained? Clearly, if the source knows the sink, the solution is simple. However, if the two do not know one another, the trust values within the social network can be used to compute a recommendation to the source regarding the trustworthiness of the sink.

Because social trust is an inherently personal concept, a computed trust value must also be personal. Thus, we do not compute a global measure of how trustworthy the sink is; rather, we use the source's perspective on the network to find paths of trust that are in turn used for making the trust computation.

TidalTrust is an algorithm for computing trust over a range of values, such as those provided by the FOAF Trust Module. It is a simple recursive algorithm: the source asks each of its neighbors for their trust rating of the sink. The source then computes a weighted average of these values, giving more weight to neighbors with higher trust ratings, and less weight to neighbors with lower trust ratings. If a neighbor has a direct trust rating of the sink, it returns that value; otherwise, the sink repeats the algorithm for *its* neighbors, and returns the weighted average that it computes. Because this algorithm is essentially a modified Breadth First Search, it runs in linear time with respect to the site of the network. More detail on the algorithm can be found at [**?**].

Previous work[**?**] has shown that the results returned by TidalTrust when it is run on both the Trust Project network and the FilmTrust network can be expected to be relatively accurate. The error varies from network to network, but is usually within 10%.

7.3 Applying Trust

These trust computations are made generally, with networks on the Semantic Web. As such, they can be integrated into a variety of applications.

There are two major projects we have undertaken to illustrate the benefit of trust ratings to the user. The first is FilmTrust, a website built on a semantic social network with movie ratings and reviews from users. On the websit, the user's movie ratings are combined with trust values to generate predictive movie ratings. We have shown that, in FilmTrust, the trust-based predictive ratings outperform other methods of generating these predictions in certain situations[**?**]. The second application is TrustMail, and email client that uses the trust rating of a message's sender as a score for the email. The trust ratings work as the complement to a spam filter by allowing users to identify potentially important messages by the trustworthiness of their sender.

Because trust and social networks are general and available publicly on the Semantic Web, there is great potential for any application to integrate this data, make computations with it, and use the results to improve the user experience.

8 Pychinko: A Rete-based RDF friendly Rule Engine

Rules continue to play a role in the Semantic Web. The numerous proposals for rule languages offered recently suggest that rules are desirable in this space, both in terms of their expressivity, and in some cases, due to their attractive computational properties. For the latter, several insights from research into rule-based expert systems in the AI community are relevant. A primary example is the Rete algorithm, developed by Forgy[**?**], which allows for efficient processing of very large rule bases.

Pychinko is a forward chaining rule engine, written in Python, that implements Rete. Its rules are expressed in the N3 language and its facts as RDF triples. The use of Rete allows it to scale far better than a similar and more popular rule engine, CWM[**?**], which uses a naive rule processing algorithm. We provide a brief overview of the Rete algorithm, and outline the features of Pychinko as well as some of its applications.

8.1 The Rete Algorithm

The basic principle of Rete (latin for *net*) is the trading of memory for speed. A typical speed inefficiency of the naive approaches to processing rules is the checking of every newly added fact (in our case, simply an RDF triple) against every antecedent in the left-hand side (LHS) of every rule, in order to determine whether the rule needs to be fired in light of the new fact. Naturally this becomes expensive as the number of rules in the knowledge base grows. The Rete algorithm improves the speed aspect of the problem by constructing a discrimination tree (or *a Rete*) that channels newly added facts only to relevant ancedents of rules, i.e. antecedents that could be matched by the fact and lead to a firing of the rule.

The Rete is built out of three components: an *alpha node*, a *beta node* and a *rule node*. It is built as follows. First, for every pattern in the LHS of a rule,

we construct a corresponding alpha node.[2] An alpha node will store facts that *match* the pattern it is associated with. This process of matching facts to their alpha nodes is called a *project operation*.

Secondly, the result of a *join operation* (same as in ordinary databases) on two alpha nodes is stored in a corresponding beta node. The resulting beta node can then be used, along with another alpha node, as input to subsequent join operations. Finally, the right-hand side (RHS) of the rule will correspond to a rule node which is always attached to the last beta node in the Rete. This is illustrated more clearly with an example. Consider the Rete corresponding to the following 2-pattern rule, represented in N3:

`{?x :parentOf ?y. ?y :parentOf ?z.} => {?x :grandparentOf ?z}`

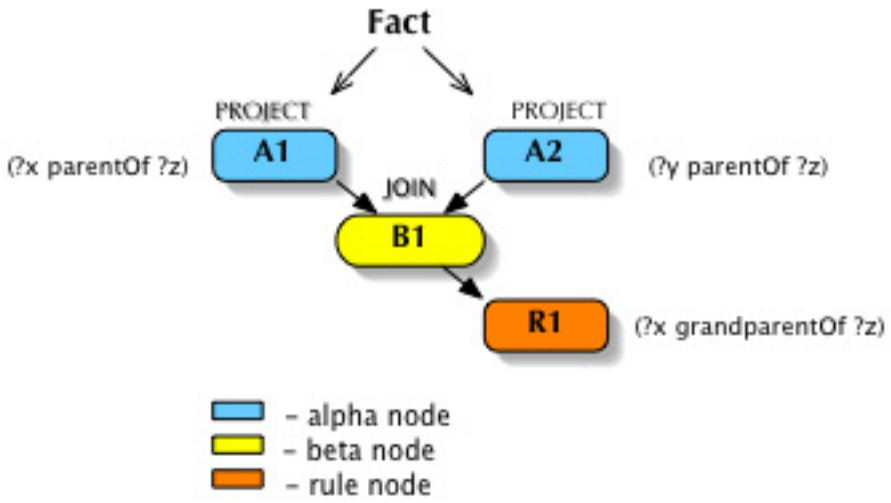

Fig. 7. Rete for a 2-pattern rule

Each arrow in the diagram describes the direction of input from one node to another. As described earlier, the first and second pattern of the LHS each correspond to an alpha node (A1 and A2, respectively), and the outcome of their join correspond to the beta node B1. The consequences of the rule (i.e. facts of the form {`?x grandparentOf ?z`} where `?x` and `?z` are properly bound) are taken care of by the rule node R1. If we were to add a third pattern to the LHS, a new beta node would be created whose inputs will be B1 and the alpha

[2] This allows us to share alpha nodes that correspond to identical patterns on the LHS of different rules. However, this is an implementation technique and is not dictated by the algorithm. We ignore it and similar optimizations in the remaining sections.

node associated with the new pattern. As usual, the results of their *join* would be stored in a new beta node.

Every newly added triple is propagated through the Rete. Starting with the project operation, it is channeled to its relevant alpha node(s) (depending on whether it matches the alpha node's associated pattern) where it is stored, assuming there are alpha nodes that match it. A series of join operations are then performed, and, assuming all are successful, the results are sent to the rule node where facts of the form described in the RHS of a rule are added to the RDF store. With these selective operations in place, it is never necessary to check a fact against every antecedent of every rule to determine a match, as done in the naive algorithms. The cost comes, of course, from having to duplicate the storage of matched facts in the nodes of the Rete.

8.2 Features, Applications and Future Work

As seen in the last section, Pychinko rules are written in N3. Alternatively, one could construct rules programmatically in Pychinko as Python objects. In addition, Pychinko provides conjunctive query over RDFlib stores and support for several useful CWM math and string builtins.

A suitable application for Pychinko would be as a backend to web sites that require a lightweight form of inference. For example, it could be used to dynamically compute closure of RDFS rules over small to mid-sized RDF stores. Similarly, it could be used be used to obtain inference for certain subsets of OWL where the overhead and completeness of a full-fledged description logic reasoner might be excessive. Such functionality is sometimes achieved by developers by writing application-specific code that generates the desired inferences from a set of OWL documents. However, in most cases, a more attractive solution is to simply write a rule that captures the inference needed, as it is reusable and less error prone. The authors of the Semantic Web toolkit 4Suite have recently made use of Pychinko for precisely these tasks in their new release, FuXi[**?**]. Finally, work is underway to integrate Pychinko into the CWM engine as an alternative faster rule processor.

9 Representing Web Service Policies in OWL-DL

To provide for a robust development and operational environment, web services are described using machine-readable metadata. This metadata serves several purposes, one of them being describing the capabilities and requirements of a service – often called the service policy. Recently, there have been many different web service policy language proposals, all of them describing languages with varying degrees of expressivity and complexity [**?**,**?**,**?**]. However, with most current proposals it is difficult to determine their expressivity and computational properties as most lack formal semantics. One characteristic of the proposed languages is that they involve policy assertions and combinations of assertions. For example, a policy might assert that a particular service requires some form

of reliable messaging or security, or it may require both reliable messaging and security. Several industrial proposals (e.g., WS-Policy [**?**] and Features and Properties [**?**]) appear to restrict them to a kind of propositional logic with policy assertions being atomic propositions and the combinations being conjunction and disjunction. By mapping the policy language constructs into a logic (e.g., some variant of first order logic) we can acquire a clear semantics for the languages, as well as a good sense of the computational aspects.

Also, if we can map the policy languages into a standardized logic, we can benefit from the tools and general expertise one expects to come with a reasonably popular standard. By mapping two policy languages into the same background formalism, we will be able to provide some measure of interoperability between policies written in distinct languages. If we are smart in our mapping, we should also be able use pre-existing reasoners for the standardized logic to do policy processing.

Mapping WS-Policy Operators to OWL Here we describe our mapping of the WS-Policy constructs from a normal form policy expression into OWL expressions. A policy in a normal form is a straightforward XML Infoset representation, enumerating each of its alternatives that in turn enumerate each of its assertions. Following is a schema outline for the normal form of a policy expression:

```
<wsp: Policy>
   <wsp:ExactlyOne>
      [ <wsp:All> [<Assertion>  </Assertion>]* </wsp:All> ]*
   </wsp:ExactlyOne>
</wsp:Policy>
```

Listing 1. Normal form of a policy expression

Policy expressions can also be represented in more compact forms, using additional operators such as `wsp:Optional`, however as shown in [**?**] the policy expressions can all be expanded to normal form. Therefore we only provide a mapping of the constructs used in a normal form policy expression: `wsp:ExactlyOne` and `wsp:All`.

First, we map policy assertions directly into OWL-DL atomic classes (which correspond to atomic propositions). Though WS-Policy assertions often have some discernible substructure, it is not key to their logical status in WS-Policy. Or rather, that substructure is idiosyncratic to the assertion set, rather than being a feature of the background formalism. So a general WS-Policy engine must be adapted to deal with their structure, if it is to do so.

Mapping `wsp:All` to an OWL construct is straightforward because `wsp:All` means that all of the policy assertions enclosed by this operator have to be satisfied in order for communication to be initiated between the endpoints. Thus, it is a logical conjunction and can be represented as an OWL intersection. Each of the members of the intersection is a policy assertion, and the resulting class

expression is a custom-made policy class that expresses the same semantics as the WS-Policy one.

Handling `wsp:ExactlyOne` is trickier, because it means that only one, not more, of the policy alternatives should be supported in order for the requester to support the policy. `Wsp:ExactlyOne` can be translated to OWL in the following way: for n different policy assertions, expressed as OWL classes themselves, `wsp:ExactlyOne` is the class expression consisting of the members of each separate policy class that do **not** also belong to another policy class. In OWL terms, it is the union of all of the classes with the complement of their pair-wise intersections.

WS-Policy Construct	OWL Expression
Wsp:All (policies A and B)	owl:intersectionOf(A B)
Wsp:ExactlyOne (policies A and B)	intersectionOf(complementOf(intersectionOf(A B)) unionOf(A B))

Table 1. Mapping of WS-Policy Constructs to OWL

WS-Policy Merge and Intersection `Merge` is the process of combining subpolicies together to form a single policy. This operation is needed because a policy might be specified in a distributed way, having its fragments defined in separate files. It is necessary to combine all these policy fragments together to form a single merged policy which could be processed further.

`Merge` works on policies already converted to normal form. The merged policy is a cartesian product of the alternatives in the first policy and the alternatives in the second policy. There is a straightforward way of doing the `Merge` operation in OWL-DL. First, we translate each of the input policies into OWL-DL as described above. Then, the merged policy is simply the *intersection* of the input policies. Thus, `Merge` also maps cleanly onto OWL-DL.

The goal of WS-Policy is to allow endpoints to specify requirements for starting a web service interaction. To achieve this goal, the `Intersection` operation compares two Web services policies for common alternatives. The interaction is possible only when both of the endpoints agree on at least one policy alternative.

`Intersection` cannot be mapped into a single OWL construct, however using our OWL mappings of the policy assertions it is not difficult to rule out the incompatible alternatives. If the policy assertions are mapped to classes, then to check whether two alternatives are equal, we need to see whether the assertions in the two alternatives are derived from the same base clases. Specifically, every assertion in the first alternative needs to be derived from the same base class with

some assertions from the second alternative, and vice-versa, for the alternatives to be compatible.

9.1 Policy Processing

One of our arguments for expressing policies using OWL was the ability to reason about policy containment - whether the requirements for supporting one policy are a subset of the requirements for another. That would allow us to be more flexible in determining whether a particular requestor supports a policy, in the cases where the requestor supports a superset of the requirements established by the policy.

In general, we get the following inferences out of the box:

1. policy inclusion (if x meets policy A then it also meets policy B; a.k.a., A rdfs:subClassOf B);
2. policy equivalence (A owl:equivalentTo B);
3. policy incompatibility (if x meets policy A then it cannot meet policy B; a.k.a, A owl:disjointWith B);
4. policy incoherence (nothing can meet policy A; a.k.a., A is unsatisfiable)
5. policy conformance (x meets policy A; a.k.a, x rdf:type A)

One futher reasoning service supported by Pellet, and integrated with Swoop [**?**], is explanations for inconsistencies [**?**], which can be used to help debug policy incompatibility, incoherence, and the like. As we add further explanation capability to our systems, this debugging power will grow.

Thus we see that with a fairly simple mapping, we can use an off the shelf OWL reasoner as a policy engine and analysis tool, and an off-the-shelf OWL editor as a policy development and integration environment. OWL editors can also be used to develop domain specific assertion languages (essentially, domain Ontologies) with a uniform syntax and well specified semantics. We can also experiment with extensions to WS-Policy, by using more expressive constructs from OWL at the policy language, as well as the assertion language, level.

Furthermore, ontology development techniques can be useful for policy development as well. Most human generate ontology develop iteratively, with specializations added to the class tree over time. Similarly, we can build up our policies from more general ones. A general policy could be very restrictive, setting tough guidelines for all of a companies policies.

If we have a similar style mapping for another policy language, we will be able to do policy analysis and integration across policy languages. We have taken the first steps in this direction with providing a translation of the Features and Properties compositors.

10 Conclusions

In this paper, we have presented tools and research projects for creating Semantic Web content with SWOOP and PhotoStuff, combining ontologies with

E-Connections, working with web services, and computing with trust in Semantic Web-based social networks. These topics illustrate both the breadth and depth of research topics on the Semantic Web, and serve as clear examples of trends in Semantic Web research.

11 Acknowledgements

This work, conducted at the Maryland Information and Network Dynamics Laboratory Semantic Web Agents Project, was funded by Fujitsu Laboratories of America – College Park, Lockheed Martin Advanced Technology Laboratory, NTT Corp., Kevric Corp., SAIC, the National Science Foundation, the National Geospatial-Intelligence Agency, DARPA, US Army Research Laboratory, NIST, and other DoD sources.